Copy No._____

Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

A Sparse Solver for VAST

M.J. Smith

Defence R&D Canada

Technical Memorandum
DRDC Atlantic TM 2002-140
September 2002

Canada

A Sparse Solver for VAST

M. J. Smith

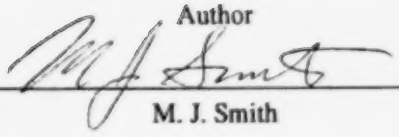
Defence R&D Canada – Atlantic

Technical Memorandum

DRDC Atlantic TM 2002-140

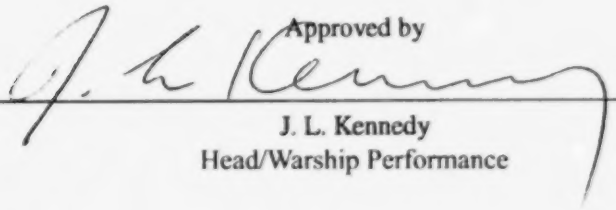
September 2002

Author



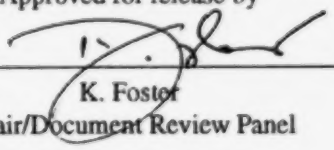
M. J. Smith

Approved by



J. L. Kennedy
Head/Warship Performance

Approved for release by



K. Foster
Chair/Document Review Panel

© Her Majesty the Queen as represented by the Minister of National Defence, 2002

© Sa majesté la reine, représentée par le ministre de la Défense nationale, 2002

106-03182/

Abstract

A sparse direct symmetric equation solver has been developed and implemented in DRDC's VAST finite element program for linear and nonlinear static analysis, eigenvalue analysis and time-history dynamic analysis. A review of sparse direct solution procedures is provided, including a discussion of fill reducing node ordering procedures. Details are given describing the sparse factorization and solution procedures implemented for very large systems of equations. The extension of the equation solver to eigenvalue and dynamic response analysis is also described. The performance of the sparse solver is compared to VAST's profile solver using several ship structure finite element models ranging in size from 7,000 to 60,000 nodes. In static response analysis the execution times for the sparse solver are typically an order of magnitude smaller than those obtained with the profile solver. In eigenvalue and dynamic response analysis the time savings are even greater. The minimum degree algorithm is found to be more effective at reducing fill-in than is the generalized nested dissection method.

Résumé

Nous avons créé et imbriqué dans VAST, le programme d'éléments finis de RDDC, un solutionneur direct de matrices d'équations creuses et symétriques qui permet d'effectuer les analyses statiques linéaires et non linéaires, l'analyse des valeurs propres et l'analyse de la réponse dynamique temporelle. Nous passons en revue les procédures directes de résolution des matrices creuses et discutons des procédures d'ordonnancement des nœuds pour simplifier la matrice. Nous expliquons en détails les procédures de factorisation et de résolution de matrices creuses utilisées pour résoudre les grands systmes d'équations. Sont aussi décrites l'utilisation du solutionneur d'équation pour l'analyse des valeurs propres et la réponse dynamique. Nous avons utilisé différents modles, par éléments finis, de structures de navire, comptant entre 7 000 et 60 000 nœuds, pour comparer le rendement du solutionneur de matrices creuses, à celui du factorisateur par méthode de profil du programme VAST. Le nouveau solutionneur a effectué les analyses de la réponse statique, en moyenne, dix fois plus rapidement que l'ancien factorisateur. Les gains de temps pour les analyses des valeurs propres et de la réponse dynamique étaient encore plus impressionnants. Nous avons découvert que l'algorithme de degré minimum était plus efficace pour simplifier le systme que la dissection emboîtée généralisée.

This page intentionally left blank.

Executive summary

Background

Finite element analysis of ship structural models is fundamental to the ongoing Improved Ship Structure Maintenance Management (ISSMM) project, which aims to provide a tool for rapid and automated risk assessment of damaged structures. A requirement of the ISSMM Software Tool (IST) is that a user be able to perform a risk assessment, which may include one or more levels of finite element analysis, within a day or less. It was determined that the performance of the VAST solver, which was selected for use with the IST, would have to be improved to meet this objective.

Principal Results

A sparse direct symmetric equation solver has been developed and implemented in DRDC's VAST finite element program for linear and nonlinear static analysis, eigenvalue analysis and time-history dynamic response analysis. The performance of the sparse solver is compared to VAST's profile solver using several ship structure finite element models ranging in size from 7,000 to 60,000 nodes. In static response analysis, sparse solver execution times are typically an order of magnitude less than those for the profile solver. In eigenvalue and dynamic response analysis the time savings are even greater. The minimum degree algorithm is found to be more effective at reducing the fill than is the generalized nested dissection.

Significance of Results

Engineering analysis of large, complex ship structure problems can now be performed more than ten times faster than was previously possible with VAST. For example, static analysis of structural detail of 10,000 nodes can now be performed in less than a minute with a standard desktop computer, whereas before it may have taken close to ten minutes. The immediate benefits are that users can now conduct many more structural analyses in a given time period, and can analyse larger, and more complex structural models than was previously possible.

Future Work

Further development may be required to allow fluid-added mass to be included in natural frequency and dynamic analysis problems. Apart from that the solver development is essentially complete, and its performance should now be sufficient to satisfy the requirements of the IST.

M. J. Smith; 2002; A Sparse Solver for VAST;
DRDC Atlantic TM 2002-140; Defence R&D Canada – Atlantic.

Sommaire

Contexte

L'analyse par éléments finis est une composante importante du projet de Gestion améliorée de la maintenance de la structure des navires (GAMSN) dont l'objectif est la production d'un outil d'évaluation rapide et automatique des risques que comportent les structures endommagées. Une des commandes pour l'outil logiciel GAMSN était la possibilité d'effectuer une évaluation des risques o interviendrait un ou plusieurs niveaux d'analyse par éléments finis, en moins d'une journée. On a établi que pour atteindre cet objectif, l'on devrait améliorer le rendement du solutionneur VAST, qui a été implanté dans l'outil logiciel.

Résultats principaux

Nous avons créé et imbriqué dans VAST, le programme d'éléments finis de RDDC, un solutionneur direct de matrices d'équations creuses et symétriques qui permet d'effectuer les analyses statiques linéaires et non linéaires, l'analyse des valeurs propres et l'analyse de la réponse dynamique temporelle. Nous avons utilisé différents modles, par éléments finis, de structure de navire, comptant entre 7 000 et 60 000 nœuds, pour comparer le rendement du solutionneur de matrices creuses, à celui du factorisateur par méthode de profil du programme VAST. Le nouveau solutionneur a effectué les analyses de la réponse statique, en moyenne, dix fois plus rapidement que l'ancien factorisateur. Les gains de temps pour les analyses des valeurs propres et de la réponse dynamique étaient encore plus impressionnants. Nous avons découvert que l'algorithme de degré minimum était plus efficace pour simplifier le systme que la dissection embotée généralisée.

Importance des résultats

On peut maintenant effectuer dix fois plus vite, qu'avec le programme VAST, l'analyse technique de grands problmes de la structure complexe de la structure des navires. On peut, par exemple, effectuer l'analyse statique de détails de structure comportant 10 000 nœuds en moins d'une minute sur un micro-ordinateur personnel, une tâche qui, précédemment, aurait exigé dix minutes. Grâce à cet algorithme, les utilisateurs peuvent effectuer, pendant un temps donné, un nombre plus élevé d'analyses structurelles. Ils peuvent également analyser des modles plus gros et plus complexes.

Travaux futurs

Des travaux supplémentaires pourraient tre nécessaires pour tenir compte dans les calculs de la fréquence naturelle et d'analyse dynamique, de la charge inertielle due à l'eau. Nonobstant ces améliorations possibles, on peut considérer que le travail sur le solutionneur est complété et son rendement devrait tre suffisant pour satisfaire aux besoins de l'outil logiciel GAMSN.

M. J. Smith; 2002; Ajout d'un solutionneur de matrices creuses au programme VAST; DRDC Atlantic TM 2002-140; RDDC Atlantique.

This page intentionally left blank.

Table of contents

Abstract i

Résumé i

Executive summary iii

Sommaire iv

Table of contents vii

List of figures viii

List of tables x

1 Introduction 1

2 Direct Solution of Positive Definite Systems 2

3 Review of Sparse Ordering Methods 3

3.1 Graph Theory Representation of Matrix Structure 4

3.2 Symbolic Factorization 6

3.3 Minimum Degree Algorithm 6

3.4 Tree Partitioning 7

3.5 One-Way Dissection 10

3.6 Nested Dissection 11

4 VAST Implementation for Linear Static Analysis 14

4.1 Node Ordering Methods 15

4.2 Symbolic Factorization 17

4.3 Sparse Assembly and Factorization 18

4.4 Sparse Solution 20

4.5 Out-of-Core Factorization and Solution 22

4.6 Sparse Data Structure 23

5 Application to Eigenvalue Analysis 25

6 Application to Linear Dynamic Analysis 27

7 Numerical Performance 28

7.1 Finite Element Models 28

7.2 Storage Requirements 33

7.3 Static Solution Performance 35

7.4 Eigenvalue Analysis Performance 37

7.5 Dynamic Analysis Performance 38

8 Conclusion 39

References 40

Annex 42

A List of symbols, abbreviations, acronyms and initialisms 42

List of figures

1 Non-zero entries of a symmetric, factorized matrix. 4

2 A symmetric matrix and its corresponding graph. 5

3 A finite element mesh and its corresponding graph. 5

4 A simple tree graph rooted at vertex r. 8

5 A tree graph with monotone ordering and its associated matrix. 9

6 A partitioned tree graph with monotone ordering and its associated matrix. 9

7 A partitioned matrix. 10

8 Mesh and matrix showing one way dissectioning with one partition. . 10

9 Mesh and matrix showing one way dissectioning with three partitions. 11

10 Mesh and matrix showing the second level of nested dissectioning. . . 12
11 Level Structure Based on the Graph in Figure 3. 14
12 Matrix Structure for Nested Dissection Ordering of Figure 3. 15
13 Program flow for sparse assembly and factorization. 16
14 Program flow for static solution in VAST. 17
15 Pseudo-code for right-looking Cholesky factorization 18
16 Pseudo-code for right-looking axpy updates 19
17 Non-zero structure of two columns. 20
18 Pseudo-code for the forward solution step. 21
19 Pseudo-code for the backward solution step. 21
20 Pseudo-code for assembly and factorization of large systems 22
21 Out-of-core forward solution of large systems. 23
22 Out-of-core backward solution of large systems. 24
23 Block pointers to panel data and index arrays, for a block with six DOF. 25
24 Longitudinal Bulkhead Detail Model. 28
25 Global Model of the HALIFAX Class Frigate. 29
26 A model of an enclosed warship mast. 30
27 HALIFAX Class Side Shell Structure in Way of Hangar. 31
28 Ring-Stiffened Cylinder Model. 32
29 Detail Model of a Ship's Hopper. 32
30 KINGSTON Class Keel Structure. 33
31 IROQUOIS Centerline Bulkhead Detail Model. 34

List of tables

1 Summary of Example Models 34

2 Matrix Size (in MB) Produced by Ordering Methods 35

3 VAST85 Static Analysis Execution Times (seconds) 36

4 VAST85 Eigenvalue Analysis Execution Times (seconds) 38

5 VAST85 Dynamic Analysis Execution Times (seconds) 39

1 Introduction

VAST (Vibration And STrength analysis) is a general-purpose finite element program developed at Martec Limited over the last three decades under sponsorship of DRDC Atlantic [1]. VAST currently provides a large selection of beam, shell and solid elements based on the state-of-the-art element technology and can perform various engineering analyses, including static, buckling, natural frequency, shock spectrum, frequency response and dynamic time-history analyses. Nonlinearities caused by arbitrarily large displacements and rotations, large strains, elastoplasticity and hyperelasticity can also be included. The VAST program has been successfully used in numerous ship structure problems and its reliability and accuracy have been well established. Recently, the VAST program has been selected as the finite element solver for the ISSMM Software Tool (IST), a DRDC Major Project now under development [2].

An important performance requirement of the IST is the ability to carry out complex evaluations of ship structures in a short period of time. The complexity may arise from the large size (i.e., number of degrees of freedom) of the finite element model, or from the large number of load cases that have to be evaluated. To achieve this goal a high degree of numerical efficiency is required from the finite element solver.

Since the 1970s VAST has relied on a profile or "skyline" type of direct solver. While it has proved to be efficient for finite element models up to a one or two thousand nodes in size, it generally gives inferior performance (compared with other FE commercial codes) when faced with more than five thousand nodes, a model size which nowadays is commonplace. In recent years, several attempts have been made to improve VAST's solution capability. These have included combining assembly and factorization operations in the existing skyline-based matrix solver [3], incorporating a commercially available matrix solver package DRMATHLIB [4], and implementing a family of iterative solvers based on the pre-conditioned conjugate gradient algorithm [5]. These refinements produced minor improvements in VAST's performance, but did not yield the significant breakthrough that was expected.

Recent studies at DRDC Atlantic with experimental equation solvers have demonstrated the benefits of sparse direct solution methods in finite-element-based ship structure analysis. Sparse direct solution methods calculate and store only the nonzero elements of the factorized coefficient matrix. In direct methods a complete factorization of the matrix is performed, thus allowing multiple load cases to be solved efficiently from a single factorization.

An experimental sparse direct solver was developed and implemented in DRDC Atlantic's object-oriented finite element program FELAM [6]. Early benchmark

results obtained with this solver indicated that as much as an order of magnitude improvement in solution time could be achieved in comparison to VAST's existing profile solver. This study also showed that performance comparable to the commercial finite element analysis program ANSYS [7] could be achieved.

Based on these findings, the decision was made to implemenent the FELAM sparse solver in VAST. This effort first required the development of an out-of-core version of the solver, i.e., one that is not limited by the system memory. The solver must also be adapted for use with VAST element formulations, and to extend its range of applicability to non-linear static, eigenvalue and dynamic analysis problems. This Memorandum provides technical details of the out-of-core sparse solver that has been implemented in VAST, and reports on the performance of the solver for typical ship structural analysis problems.

2 Direct Solution of Positive Definite Systems

A comprehensive review of sparse matrix solution methods is beyond the scope of this study. Entire books have been devoted to the subject (see [8] – [10]); and the properties of positive definite systems have been discussed thoroughly in linear algebra textbooks (see [11] – [13]). Only a brief summary of the problem is provided here.

Static equilibrium of a structural system subject to finite element discretization is expressed by the linear algebraic equation

$$Ku = f \tag{1}$$

where K is a $n \times n$ symmetric positive definite stiffness matrix of the structure, f is a $n \times 1$ load vector, and u is a $n \times 1$ displacement vector; n is the total number of degrees of freedom, which in most ship structures problems is equal to six times the number of nodes (three displacements and three rotations per node). When n is large, the number of zero entries in K is typically much larger than the number of non-zero entries. In other words, K is sparse. The distribution of non-zeroes in the stiffness matrix depends on the connectivity of the finite element model and the particular ordering of the finite element nodes in Equation (1). The diagonal entries of K must all be positive in properly formulated models.

Direct solution methods calculate u for a given f by first decomposing the stiffness matrix to the form $K = LDL^T$, where L is a lower triangular matrix with unit diagonals and D is a diagonal matrix. (This is a slightly more general form of the usual Cholesky decomposition LL^T). The decomposition is referred to as the

factorization step, and can be performed using the various forms of the Cholesky algorithm [8]. A solution step is then performed in which u is obtained by solving the following three systems in sequence

$$Lv = f \qquad Dw = v \qquad L^T u = w \tag{2}$$

Solving each of these systems for large n is relatively inexpensive compared with the factorization step. But if multiple load vectors are present, entailing the repeated solution of (2), the cost of the solution step can become quite comparable to, or even greater than, that of the factorization step.

3 Review of Sparse Ordering Methods

In sparse direct methods, only the non-zero elements in K and L are stored during the factorization and solution steps. The non-zeroes may be distributed in any symmetric fashion, and need not be clustered around the diagonal as in profile or band methods [8].

An important consequence of the factorization step is the *fill-in* that occurs in the triangular factor L. This is where non-zero entries are introduced in L at column and row positions where zeroes exist in K. Fill-in is the by-product of the row and column operations performed during Cholesky factorization. Figure 1 shows the non-zero structure in the lower triangle of a factorized symmetric matrix L. In this figure, $\times$ represents the non-zeroes in the original matrix K, and $+$ are fill-in entries. A sparse data structure set up to store just the non-zeroes of K will not be sufficient for storing the non-zeroes of L unless no fill-in occurs at all. This is contrary to profile or band methods where the fill-in is always confined to the profile or band of the original matrix.

The amount of fill-in strongly depends on the node ordering; orderings that produce large fill-in will be expensive to solve and should be avoided. The optimal ordering would be one that minimizes fill-in, but in practice it is not possible to find such a minimal ordering. Instead, sparse ordering methods tend to find *good* orderings, i.e., ones that produce moderate fill-in, but generally more than the theoretical minimum. The ordering methods tend to be heuristic in nature, exploiting simple rules that are known to reduce fill-in.

Sparse ordering methods are typically based on concepts from graph theory since these provide a means of symbolically representing the factorization process. While an extensive discussion of graph theory is unnecessary, a few concepts will be introduced as a prelude to discussion of specific ordering methods.

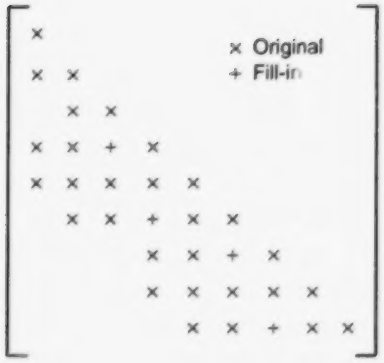


Figure 1: *Non-zero entries of a symmetric, factorized matrix.*

3.1 Graph Theory Representation of Matrix Structure

Figure 2 shows the non-zero structure of a symmetric matrix and its corresponding representation as a graph. The graph consists of *vertices* (shown as numbers in circles) and *edges* (shown as lines connecting vertices). The vertices represent the diagonal entries of the matrix and are numbered according to their row or column position. The edges represent the coupling of two vertices by the off-diagonal matrix entries. Because the matrix is symmetric, edges have no preferred direction and the graph is called an *undirected* graph. For nonsymmetric matrices, the edges have a preferred direction since the coupling of vertex i to vertex j does not necessarily imply the coupling of j to i. In this study only symmetric matrices and undirected graphs are considered.

A pair of vertices connected by an edge is said to be *adjacent*. The *adjacencies* of a vertex u, denoted by Adj(u), are all the vertices that share an edge with u, not including itself. The *degree* of a vertex is the number of its adjacent vertices.

A *path* exists between two vertices u and v when a set of vertices $u = x_0, x_1, \ldots, x_{m-1}, x_m = v$ exists such that each consecutive pair x_{i-1}, x_i, $i = 1 \ldots m$, is adjacent. A graph is said to be *strongly connected* if a path exists between every pair of vertices. A *cycle* is a path consisting of at least four vertices in which the first and last vertices are the same; it can be identified as a loop in the graph.

A direct relationship exists between a finite element mesh and the graph of its assembled coefficient matrix. A finite element node can be represented in a graph as a vertex, while the elements of the mesh define the edges coupling the vertices. A set of nodes belonging to the same element will appear in the graph as a group

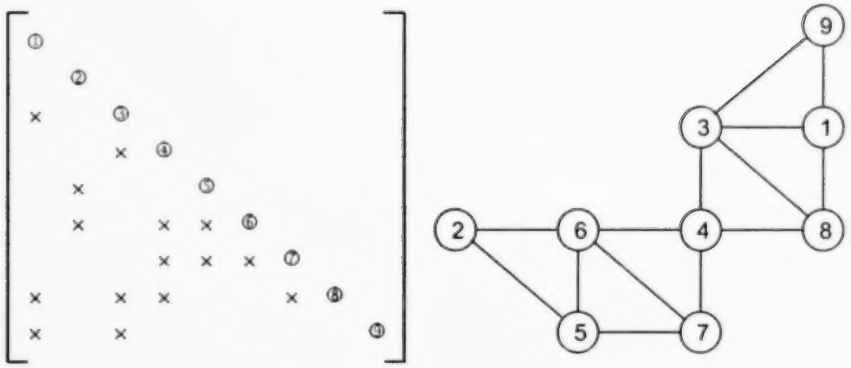


Figure 2: *A symmetric matrix and its corresponding graph.*

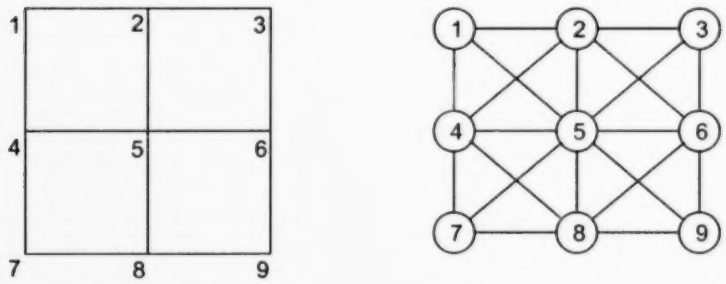


Figure 3: *A finite element mesh and its corresponding graph.*

of vertices mutually connected by all possible edges. Thus vertices belonging to the same element are all adjacent to one another, and such a set of vertices is said to form a *clique*. Figure 3 illustrates a simple finite element mesh and the graph resulting by forming cliques in the elements.

If finite element nodes have more than one degree of freedom associated with them, their graphical representation is unchanged. This can be understood from Figure 1 which gives the symmetric coefficient matrix associated with the finite element mesh in Figure 3. Each $\times$ now represents a $d \times d$ block of entries, where d is the number of degrees of freedom per node. The $d \times d$ blocks are in general densely populated with non-zero entries, and so what sparsity exists in the matrix can be fully accounted for at the nodal level. Sparse ordering methods can therefore be discussed assuming a single degree of freedom per node. This approach implicitly makes use of the *supervertex* concept, in which matrix columns with identical non-zero patterns are symbolically lumped together [8].

3.2 Symbolic Factorization

Cholesky factorization works in a manner similar to the familiar Gaussian elimination process, whereby non-zero off-diagonal elements are progressively eliminated starting from the left-most column by performing a sequence of algebraic operations. These operations affect entries in columns to the right of the eliminated columns. If the matrix is viewed symbolically as a graph, eliminating a column of the matrix is akin to eliminating a vertex in the graph. When a vertex i is eliminated, the graph is transformed according to the following process:

Step 1 Remove vertex i and all of its edges from the graph.

Step 2 Add edges such that the vertices in Adj(i) form a clique.

In terms of matrix operations, Step 1 corresponds to the zeroing of the off-diagonals in column i, whereas Step 2 corresponds to the filling-in of the columns to the right of column i.

Thus the factorization of a matrix can be represented symbolically as a series of graph transformations. Denoting G as the graph of the original unfactorized matrix, the progressive elimination of columns 1 to n produces a series of *elimination graphs* denoted as $G^2, G^3, \ldots G^n$, where G^{i+1} is the elimination graph resulting from the elimination of the first i vertices. The graph $G^{i+1}, i > 0$ is generated from G^i by applying the above two step transformation.

The adjacencies of the ith vertex in the factorized matrix are obtained directly from the elimination graph G^{i+1}. The sequence of elimination graphs, $G^2, G^3, \ldots, G^n$, therefore gives the adjacencies for all the vertices in the graph, and hence provides the entire non-zero structure of the factorized matrix.

The significance of symbolic factorization is that the non-zero structure caused by eliminating the vertices in a particular order can be predicted in advance of forming the coefficient matrix. This is very important because it allows the sparse data structure and storage required for a factorized coefficient matrix to be predicted and allocated prior to assembling the matrix. Otherwise, separate data structures and storage for the original and factorized coefficient matrices would be needed. Furthermore, the analyst would be faced with having to allocate and maintain the sparse data structure for the factorized matrix while the factorization was taking place.

3.3 Minimum Degree Algorithm

With the basic concepts of graph theory and symbolic factorization described in the preceding sections, the order in which vertices are eliminated can now be consid-

ered.

The simplest ordering method for sparse factorization is the minimum degree (MD) algorithm, originally developed by Rose [14]. The MD algorithm specifies that at any stage of the elimination, the next vertex eliminated should be the one with the minimum degree. The first minimum-degree vertex is numbered '1' in the new ordering, with the others being renumbered sequentially as they are selected and symbolically eliminated. The degrees of the vertices are taken from the most recent elimination graph, and therefore continually updated during the symbolic factorization process.

The MD algorithm provides good orderings because at each stage of the process, eliminating a vertex with the smallest degree generally produces the smallest amount of fill-in. Thus the method is sometimes referred to as a local minimization or "bottom-up" strategy because it tries to minimise the fill-in for each vertex separately, in piecemeal fashion, without considering the total fill-in produced by the ordering.

It often occurs that at a given elimination step more than one vertex will have minimum degree, and thus a tie-breaking strategy is needed. Unfortunately, no tie-breaking strategy has yet been found that is better than an arbitrary selection from among the minimum degree nodes [15].

The MD algorithm can be improved through a procedure known as *mass elimination* of indistinguishable vertices. Two vertices are said to be indistinguishable in any given graph if they have identical *neighbourhood sets*, where a neighbourhood set of a vertex is defined as the union of itself and its adjacencies. Thus, vertices u and v are indistiguishable if $u \cup \text{Adj}(u) = v \cup \text{Adj}(v)$. A set of indistinguishable vertices can be eliminated in any order without changing the fill-in pattern. When a substantial number of vertices are indistinguishable, it is more efficient to mass-eliminate the vertices than to eliminate them one at a time. The trade-off is that some work has to be done to identify the vertices that are indistinguishable.

3.4 Tree Partitioning

A *tree* is a special type of graph in which a unique path exists between each pair of vertices. Another way of defining a tree is to say that it is strongly connected and contains no cycles. An example tree graph is shown in Figure 4. A tree graph is said to be *rooted* at the uppermost vertex r. The unique path between a vertex and the root defines an ancestor-descendent relationship among the vertices. For a path extending from the root labelled as $v_0 = r, v_1, v_2, \ldots v_m$, vertex v_i is said to be the descendent of vertices $v_j, j < i$, and the ancestor of vertices $v_j, j > i$. Vertex v_{i-1}

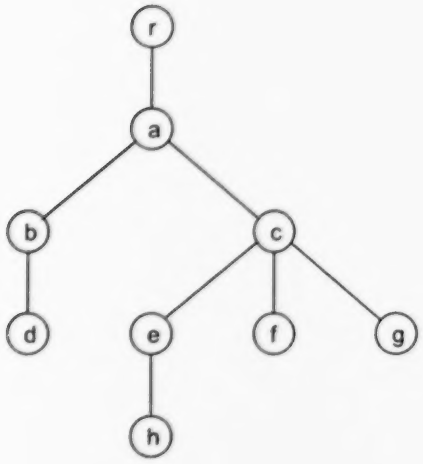


***Figure 4:** A simple tree graph rooted at vertex r.*

is the *parent* of v_i, while v_{i+1} is its *offspring*. Thus, in Figure 4, e is the parent of h and the offspring of c. The ancestors of g are r, a, and c.

The tree graph is *monotone ordered* if the vertices are labelled before their parents. The significance of this is that a symmetric matrix associated with a monotone-ordered tree graph will experience no fill-in during factorization. Figure 5 shows the tree graph of Figure 4 in monotone ordering, and the corresponding non-zero structure. It can be verified that no fill-in occurs during the factorization of this matrix.

Apart from some simple beam and bar structures, few finite element structural models of practical interest have graphs in the form of trees. But if a matrix is partitioned such that one or more vertices are grouped together to form composite vertices, then in general it is possible to choose a partitioning such that the graph formed by the composite vertices is a tree. An example of a tree partitioning is given in Figure 6, where the graph from Figure 2 is partitioned into four composite vertices. Partitioning a graph in this manner partitions the associated matrix into blocks of rows and columns. Arranging the composite vertices in monotone order ensures that fill-in is confined to non-zero blocks established by the tree partitioning. It also ensures that no more than one off-diagonal non-zero block will appear below each diagonal block. Figure 7 shows the partitioned matrix associated with the tree partitioning in Figure 6. Because the composite vertices are in monotone order, no fill-in can occur in the shaded region.

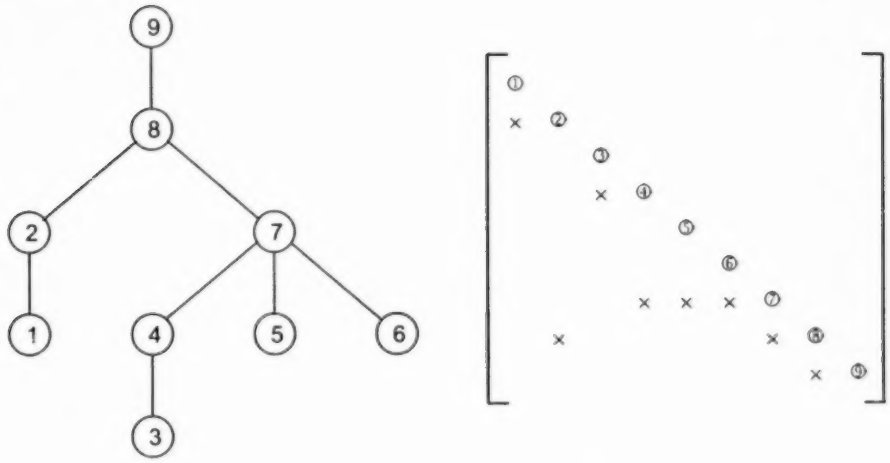


Figure 5: *A tree graph with monotone ordering and its associated matrix.*

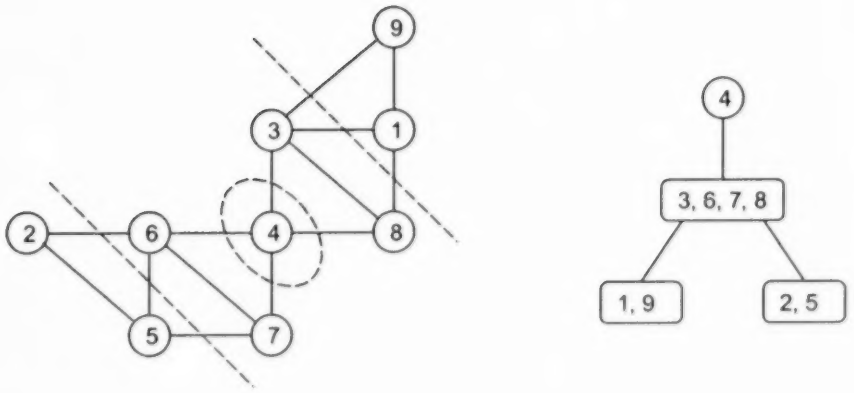


Figure 6: *A partitioned tree graph with monotone ordering and its associated matrix.*

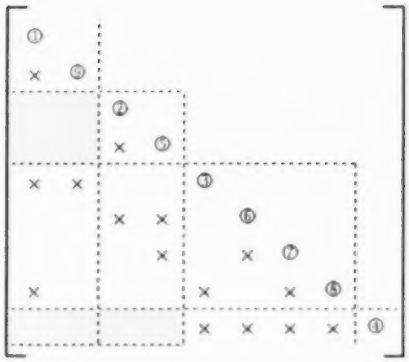


Figure 7: *A partitioned matrix.*

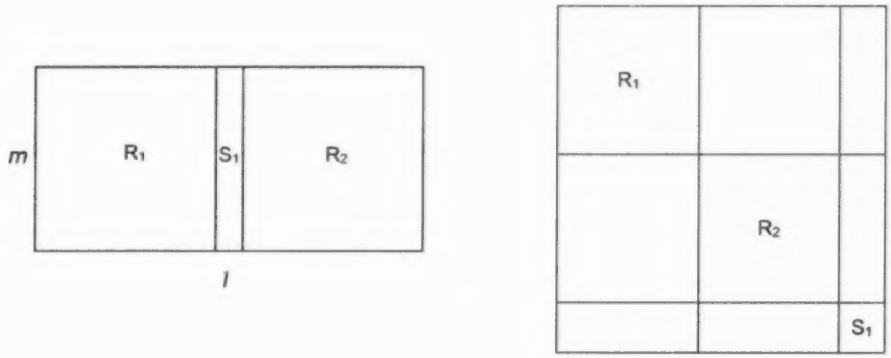


Figure 8: *Mesh and matrix showing one way dissectioning with one partition.*

3.5 One-Way Dissection

Consider the ordering of a two dimensional mesh of size $l \times m$, with $l \gg m$, shown in Figure 8. If the mesh is split in two, such that the two components R_1 and R_2 are numbered first and second, and the separator S is numbered last, the resulting structure of the coefficient matrix is shown in Figure 8. Two decoupled diagonal blocks of approximate size $lm/2 \times lm/2$ are introduced for the two components R_1 and R_2. The separator introduces a small diagonal block of size $l \times l$, and two off-diagonal blocks of size $l \times lm/2$. The rest of the matrix is zero and remains zero during factorization; no fill-in occurs outside of the diagonal blocks and blocks coupling to the separator.

If instead, σ evenly-spaced separators are introduced, and nodes are ordered such that the components R_1 through $R_{\sigma+1}$ are numbered first, followed by separators

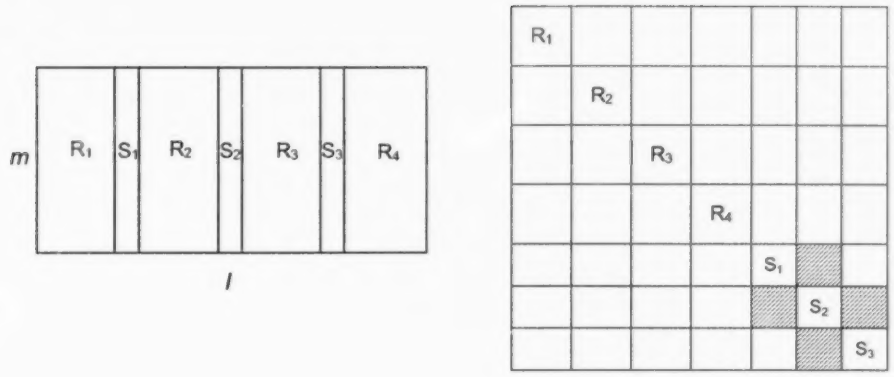


Figure 9: *Mesh and matrix showing one way dissectioning with three partitions.*

S_1 through S_σ, the non-zero block structure can be refined further. The case for $\sigma = 3$ is shown in Figure 9. Fill-in occurs now within the shaded blocks, and also in the cross-hatched blocks indicated. The fill-in that occurs in both the shaded and the cross-hatched blocks is not complete, and their sparseness after fill-in can be subsequently determined by symbolic factorization.

For a given rectangular $l \times m$ mesh, it is possible to determine optimal values for σ that produce either a minimum storage, or a minimum number of operations for either the factorization or solution steps [8]. Thus the selection of separators is easily automated for this two-dimensional configuration. Identifying separators for more general meshes is done using *level structures*, in which vertices are partitioned into a number of hierarchical levels. Any level, other than the top or bottom level, may serve as a separator between levels above and levels below. Good orderings can be obtained with one-way dissection when separators can be kept small, and when a long level structure (i.e., one with many levels) can be identified.

3.6 Nested Dissection

Another approach to ordering is nested dissection, in which successive levels of dissection are applied to a matrix until it can be dissected no further. The process can be discussed in terms of a square $m \times m$ mesh, in which case the first level of nested dissection shown in Figure 8. In the next level of dissection, the two components R_1 and R_2 are subdivided by finding separators S_1^2 and S_2^2. This is indicated in Figure 10. This produces four components R_1^2, R_2^2, R_3^2, and R_4^2, where the superscript 2 indicates the level of dissection.

In nested dissection orderings, the nodes of the decoupled components are labelled

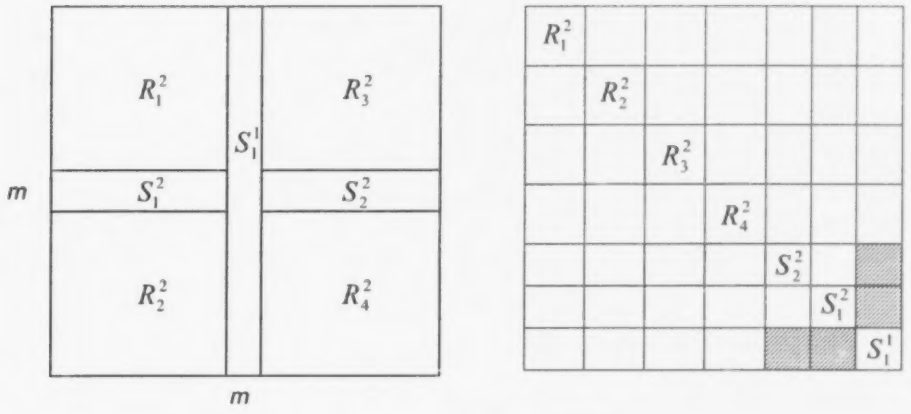


Figure 10: *Mesh and matrix showing the second level of nested dissectioning.*

first, followed by the nodes of the separators in the reverse order in which they were found. The non-zero structure of the matrix produced by this ordering will be as shown in Figure 10. No fill-in will occur outside of the shaded and cross-hatched blocks. As with one-way dissection, the exact non-zero structure within the blocks can be subsequently determined with symbolic factorization.

The dissection process can be continued in a similar manner until the components are reduced to any desired minimum size. The term *generalised nested dissection* refers to a continuation of the recursive process until the components can no longer be dissected [9]. Alternatively, the recursive process can be halted at any point, and the nodes of the individual components and separators can then be locally ordered using MD or a profile method. Methods of this sort are sometimes called *incomplete nested dissection*.

Nested dissection has been shown to be the optimal ordering method for regular $m \times m$ square meshes [8]. Indeed, in this special case it can be shown to produce the minimum-fill ordering. It is less effective for a rectangular $l \times m$ mesh with large aspect ratio such that $l \gg m$; and in this case MD or one-way dissection may be better. For an arbitrary structural mesh in three-dimensions, the effectiveness of nested dissection can vary, and depends largely on the quality of the separators that can be found.

Both one-way and nested dissection work by subdividing a mesh into a set of disconnected components. Ashcraft and Liu [19] have described a more generalised approach called *multisection* in which the two dissection approaches appear as special cases. Their work suggests that the best ordering approach for a given problem is a hybrid approach, in which a mesh is decomposed into components using a

dissection method, and then the components and separators are ordered using a mixture of approaches.

A heuristic algorithm for generalized nested dissection (GND) has been provided by George and Liu [8]. The algorithm uses level structures of connected components within a graph to find separators that roughly bisect the components. Before describing the algorithm, it is first necessary to define a level structure. A level structure rooted at a vertex x is a partitioning of the vertices $\mathcal{L}(x)$ such that

$$\mathcal{L}(x) = \{L_0(x), L_1(x), \ldots, L_{\ell(x)}(x)\}$$

where $L_i(x)$ are sets of vertices defined as follows:

$$L_0(x) = \{x\} \qquad L_1(x) = \text{Adj}(L_0(x))$$

$$L_i(x) = \text{Adj}(L_{i-1}(x)) - L_{i-2}(x), \quad i = 2, 3, \ldots, \ell(x)$$

In other words, the vertices in a level consist of the adjacencies of the vertices in the previous level, minus any that have appeared in a higher level. The exception of course is the first level which contains only the single vertex x. The *length* of a level structure is $\ell(x)$. Usually, one wants a level structure rooted at a *pseudo-peripheral* vertex, which can be roughly defined as a vertex on the periphery of the graph or mesh. A well defined algorithm exists for identifying pseudo-peripheral vertices, but it is not necessary to describe it here. For the graph depicted in Figure 3, vertex 9 is a peripheral vertex. It is easy to verify that the level structure at this vertex will be as depicted in Figure 11.

The George and Liu algorithm for GND goes as follows:

1. Generate an undirected graph G for the mesh. Set $R = \emptyset$.

2. For a connected component C within G, construct a level structure rooted at a pseudo-peripheral vertex r.

3. Find a separator S for the level structure. If $\ell \leq 2$, $S = C$ and go to Step 4. Otherwise, choose the separator as $S = L_j, j = (\ell + 1)/2$, and then remove from S any vertices that do not have adjacencies in L_{j+1}.

4. Set $R \leftarrow R \cup S$. If $|R| = n$, the total number of vertices, then go to Step 5. Otherwise remove vertices in S from G along with their incident edges and go to to Step 2.

5. Obtain node ordering by reversing the order of the vertices in R.

If this algorithm is applied to the level structure in Figure 11, the first iteration selects vertex 4 in level L_2 as the first separator. (Vertex 8 is excluded because

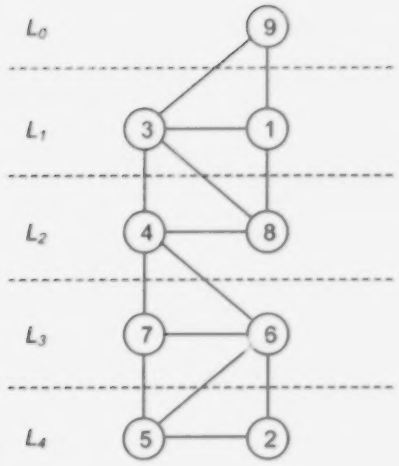


***Figure 11:** Level Structure Based on the Graph in Figure 3.*

it has no adjacencies in L_3.) The graph is thereby divided into two disconnected components: $\{9, 3, 1, 8\}$ and $\{7, 6, 5, 2\}$. The two following iterations of GND will identify the two separators $\{3, 1\}$ and $\{5, 6\}$, which divides the two components into the four disconnected components $\{9\}$, $\{8\}$, $\{7\}$, and $\{2\}$. Thus the final ordering is $\{2, 7, 8, 9, 5, 6, 3, 1, 4\}$ and the structure of the assembled and factorized matrix is given in Figure 12.

4 VAST Implementation for Linear Static Analysis

The sparse solver described here was first developed and tested for the object-oriented finite element code FELAM [6], in which it was used for linear static response analysis. It was then implemented in VAST version 8.3 for both linear and nonlinear static analysis, as described by Jiang [20]. The VAST implementation required changes to the solver code specifically related to VAST. These included:

1. Use of the entire VAST element library
2. Assembly of element matrices from VAST binary format
3. Inclusion of VAST external and multi-point constraints
4. Checking and correcting for linear dependencies in the assembled matrix

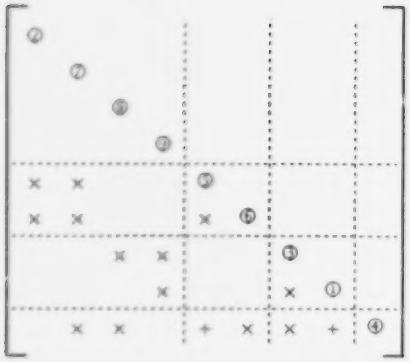


Figure 12: *Matrix Structure for Nested Dissection Ordering of Figure 3.*

5. Simultaneous solution of multiple load vectors

6. Storage of solution vectors in VAST format

More recently, the solver implementation has been extended for eigenvalue type problems (natural frequency and linear buckling analysis) and for linear dynamic response (transient) analysis. This implementation has been completed in version 8.5 of VAST.

Throughout the VAST implementation, the original object-oriented structure of the solver code has been retained. Figures 13 and 14 show the program flow for the sparse factorization and static solution phases, respectively, as they are implemented in VAST. The following sections describe the functioning of principal procedural steps.

4.1 Node Ordering Methods

Of the four node ordering methods outlined in the previous chapter, generalized nested dissection (GND) is perhaps theoretically the most attractive since it offers a top down approach to finding low fill-in orderings. On the other hand, the minimum degree (MD) algorithm has been found in practice to be reliable, gives satisfactory orderings in all problems that have been tested, and is relatively easy to implement.

Both the MD and GND ordering methods have been implemented in the most recent version of the sparse solver. The MD algorithm used is integrated with the symbolic factorization algorithm, which is described in the next section. Recent improvements have included the use of mass elimination of indistiguishable nodes, as was described in Section 3.3. The GND algorithm is based on the heuristic

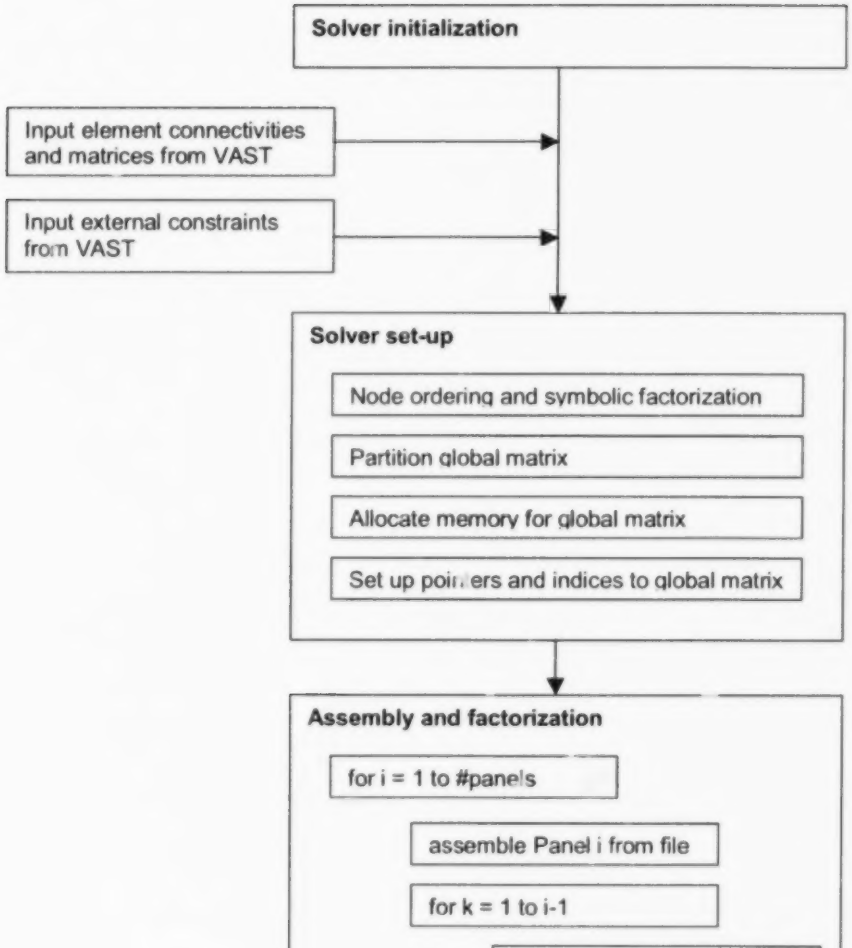

Solver initialization
Input element connectivities and matrices from VAST
Input external constraints from VAST
Solver set-up
Node ordering and symbolic factorization
Partition global matrix
Allocate memory for global matrix
Set up pointers and indices to global matrix
Assembly and factorization
for i = 1 to #panels
assemble Panel i from file
for k = 1 to i-1
update Panel i from Panel k
factorize Panel i
store Panel i on file

Figure 13: *Program flow for sparse assembly and factorization.*

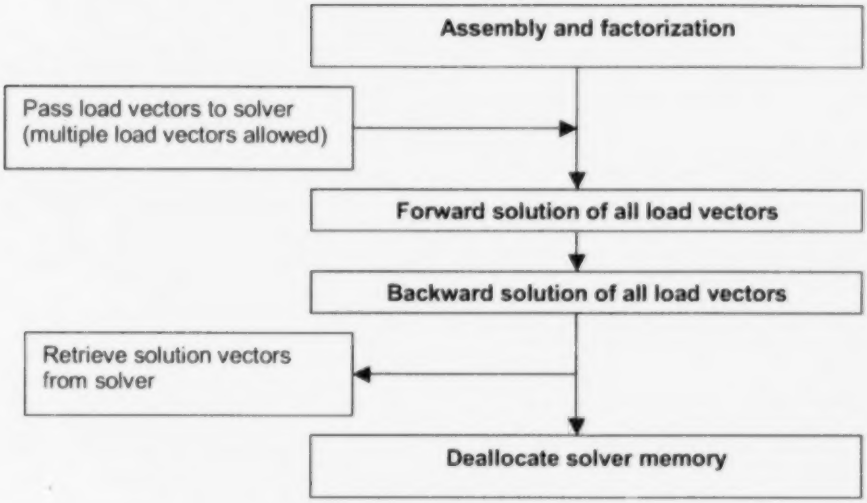

Assembly and factorization
Pass load vectors to solver
(multiple load vectors allowed)
Forward solution of all load vectors
Backward solution of all load vectors
Retrieve solution vectors
from solver
Deallocate solver memory

Figure 14: *Program flow for static solution in VAST.*

method outlined in Section 3.6. The performance of the two ordering methods is compared in Section 7.2.

4.2 Symbolic Factorization

Starting from the mesh connectivities, a data structure is created to represent the graph of the mesh. The edges shared by a vertex are represented as an ordered list of adjacent vertices. With this representation, adding or deleting edges is equivalent to adding or subtracting adjacencies. Expandable vector objects from the Standard Template Library of C++ [16] are used to store the adjacencies, so that insertion and deletion operations can be done efficiently.

Once the graph of the mesh is established, symbolic factorization is performed using the method described in Section 3.3. The transformation of the original graph to the series of elimination graphs $G^2 \ldots G^n$, is performed using an *explicit* model of the graph, in which the two-step procedure given in Section 3.2 is performed directly on the adjacency lists. This is different from the *implicit* model, which uses *reachable sets*, and the *quotient* model, which uses *quotient graphs*, to represent the elimination process [8]. It is not necessary here to describe how implicit and quotient models work in detail; but their principal advantage is that by using them, the amount of memory required to contain the elimination process is predictable, and therefore the operations can be performed using an array of fixed size. But since the C++ implementation being used here can make efficient use of expandable

```
factorize(firstColumn, lastColumn) :
    for i = firstColumn : lastColumn
        columnDiv(i)
        for j = i + 1 : lastColumn
            columnMod(j, i)
        end
    end

columnDiv(i) :
    L_:i = L_:i / D_i

columnMod(j, i) :
    if L_ji ≠ 0
        D_j = D_j − D_i L_ji^2
        L_:j = L_:j − L_:i D_i L_ji
    end
```

Figure 15: *Pseudo-code for right-looking Cholesky factorization*

arrays for the adjacency lists, an explicit approach should also be effective. This avoids having to use the more complex algorithms needed for implicit and quotient models.

4.3 Sparse Assembly and Factorization

The outer-product, or "right-looking" form of Cholesky factorization is convenient for large matrices that use sparse column storage of the lower triangle [17]. The algorithm for computing the factor L is given by the pseudo-code Figure 15, where the diagonal entries D and the off-diagonal entries L are computed separately. Here $L_{:i}$ denotes the off-diagonal portion of the ith column of L, L_{ji} is the (j,i) entry of L, and D_i is the ith diagonal.

The factor L is initialized to the lower triangle of the assembled K matrix, and D is initialized to the diagonal of K. Efficient use of memory is achieved by allocating a data structure for L following the symbolic factorization step, and then assembling into it the off-diagonal symmetric lower half of K. This avoids the inconvenience of having to use separate blocks of memory for K and L. But it is convenient to store D and L separately, and this can be done without any significant loss of efficiency.

In Figure 15 it can be seen that the function **columnMod** will use the bulk of the computational effort. This function performs the updates of the diagonals and columns of L required for the right-looking version of the Cholesky algorithm. The updates of the diagonals are straight forward and can be implemented efficiently.

```
rightLookingAxpy(a, x, y, startPos) :
    yPos = 0
    xMax = length(x)
    for xPos = startPos : xMax
        while iy(yPos) < ix(xPos)
            yPos = yPos + 1
        end
        y(yPos) = y(yPos) + a x(xPos)
        xPos = xPos + 1
        yPos = yPos + 1
    end
```

Figure 16: *Pseudo-code for right-looking axpy updates*

The updates of the columns of L are complicated by the fact that the columns are in sparse format, where each column is represented by a double precision array for the column entries, and an integer array holding the row indices of the entries.

An efficient algorithm for performing the column updates is given in Figure 16. The algorithm performs the vector "axpy" operation $y = y + ax$, where x and y are sparse vectors representing columns of L. When updating column j with column i the scalar constant a is $-D_i L_{ji}$. Vectors ix and iy hold the corresponding row indices for the entries of x and y, respectively. The variables *xPos* and *yPos* mark the positions being referenced in the x and y vectors, respectively. The axpy operation is performed only on entries of the x and y where corresponding indices match up.

The non-zero structure of two representative columns of L is shown in Figure 17. Column j is updated by column i only if $L_{ji} \neq 0$. If this is the case then two facts about the non-zero structure can be deduced. First, the entries of $L_{:i}$ with indices less than or equal to j will not affect the j column, and can be automatically skipped in the axpy operation. Second, for each non-zero entry in $L_{:i}$ with an index greater than j, there must be a corresponding non-zero entry in $L_{:j}$. This fact directly follows from the fill-in principle. The converse of this rule, i.e., that for each entry in $L_{:j}$ with an index greater than j, there will be a corresponding entry in $L_{:i}$, is *not* true.

These two facts are exploited in the **rightLookingAxpy** algorithm. A parameter *startPos* is used to hold the starting reference point for x and ix. This allows the irrelevant entries at the start of these vectors to be skipped. This parameter is updated and maintained throughout column updates so that minimal searching is needed to determine *startPos*.

With these preliminary entries skipped, the search for matching indices begins. If

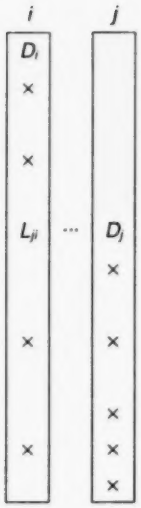


Figure 17: *Non-zero structure of two columns.*

a match is found, the axpy operation is performed on the entries. If the indices do not match, the entries are skipped. Because of the matrix structure imposed by the fill-in, the only possible mismatch that can occur is $iy(yPos) < ix(xPos)$, in which case *yPos* is advanced. The condition $iy(yPos) > ix(xPos)$ never has to be checked, and this leads to a significant saving. The search for matching indices continues until the last entry of x, denoted by the position *xMax*, is reached.

4.4 Sparse Solution

Following the factorization step, solution of the three algebraic systems in (2) is performed. The lower triangular and diagonal systems can be solved together, so in effect only two triangular systems have to be solved. The pseudo-code for these operations is given in Figures 18 and 19. In the interests minimizing memory usage, the load vectors f and the solution vectors u share memory, the solution vectors overwriting the load vectors as they are calculated.

In the forward solution the bulk of the computational effort takes place in the innermost for-loop, which consists of a partial vector axpy operation on the vector x using the columns of L. In the backward solution the bulk of the effort is again in the innermost for-loop, which is the inner product of x with a column of L. In both of these cases the vector x is in dense format (i.e., not sparse). The axpy and inner product operations can therefore be efficiently implemented in a straight forward manner, and it is not necessary to described them in detail.

```
forwardSolve(firstColumn, lastColumn, x) :
    for i = firstColumn : lastColumn
        if x_i ≠ 0 then
            x_i = x_i / D_i
            for j = i + 1 : n
                x_j = x_j − L_ji x_i
            end
        end
    end
```

Figure 18: *Pseudo-code for the forward solution step.*

```
backwardSolve(firstColumn, lastColumn, x) :
    for i = lastColumn : firstColumn
        s = 0
        for j = i + 1 : n
            s = s + L_ji x_j
        end
        x_i = x_i − s
    end
```

Figure 19: *Pseudo-code for the backward solution step.*

```
factorizeLarge :
    for k = 1 : numPanels
        assemblePanel(k)
        for m = 1 : k − 1
            updatePanel(k, m)
        end
        factorize(firstColumn(k), lastColumn(k))
        if numPanels > 1 then
            storePanel(k)
        end
    end

updatePanel(k, m) :
    for i = firstColumn(m) : lastColumn(m)
        readColumn(i)
        for j = firstColumn(k) : lastColumn(k)
            columnMod(j, i)
        end
    end
```

Figure 20: *Pseudo-code for assembly and factorization of large systems*

4.5 Out-of-Core Factorization and Solution

When the storage required for the factor L exceeds the available memory, an out-of-core approach to the factorization is required. A panel factorization scheme is used in which the matrix is broken up into groups of adjacent columns (called "panels"), each of which is small enough to fit in memory. This turns out to be a convenient arrangement when using column storage. If the matrix is small enough to be solved using a single panel, the procedure used is equivalent to in-core factorization and solution.

The out-of-core algorithm **factorizeLarge** developed for the VAST solver is a modification of one given by Rothberg and Schreiber [18] and is shown in Figure 20. The function **assemblePanel**(k) assembles the columns for the kth panel. Function **updatePanel** is then called for each of the $k-1$ previously factorized panels. This performs column operations on the current panel required by the previous factorization steps. The column updates are identical to those defined in the **columnMod** procedure in Figure 15.

In the function **updatePanel**(k, m), the functions *firstColumn* and *lastColumn* return the first and last column indices for a given panel. The columns of the previ-

```
forwardSolveLarge :
    for k = 1 : numPanels
        if numPanels > 1
            retrievePanel(k)
        endif
        for m = 1 : numLoadVectors
            x = loadVector(m)
            forwardSolve(firstColumn(k), lastColumn(k), x)
            solnVector(m) = x
        endfor
    endfor
```

Figure 21: *Out-of-core forward solution of large systems.*

ously factorized panels are read from disk individually, and are then used to update the columns of the current panel k. Thus it is possible to carry out the factorization with no more than a single panel and a single column stored in memory at once.

The function call **factorize** in Figure 20 invokes the right-looking procedure in Figure 15 to factorize panel k. When the panel is factorized it is written to disk, unless there is only one panel in which case the factorization and solution can be performed in-core and no accessing of the disk is required.

Extension of the solution methods to out-of-core computation is straightforward, and is given in Figures 21 and 22. These routines use the forward solution method **forwardSolve** and a backward solution method **backwardSolve**, given in Figures 18 and 19. The out-of-core methods have been extended for use with multiple load and solution vectors. The function **loadVector**(m) is used to denote the retrieval of the mth load vector from disk or memory; **solnVector**(m) denotes the storage of the mth solution vector. Regardless of the number of solution vectors to be calculated, each panel only has to be retrieved twice in the solution algorithm given, once in the forward solution and once in the backward.

4.6 Sparse Data Structure

Two one-dimensional arrays are allocated for holding the double-precision off-diagonal data and integer indices defining the columns of the matrix. Data for the matrix diagonal is stored separately, as described below. The data and indices for only one panel are stored in memory at a time. Thus, these arrays will be referred to as the *panel arrays*. The amount of memory required for each panel is determined in advance using the results of the symbolic factorization and the amount of available memory. Since the allocation is only performed once, the panel arrays must be made large enough to hold the largest panel, but no larger.

```
backwardSolveLarge :
    for k = numPanels : 1
        if numPanels > 1
            retrievePanel(k)
        endif
        for m = 1 : numLoadVectors
            x = solnVector(m)
            backwardSolve(firstColumn(k), lastColumn(k), x)
            solnVector(m) = x
        endfor
    endfor
```

***Figure 22**: Out-of-core backward solution of large systems.*

With the panel arrays allocated, pointers to each column in the matrix are established so that the individual columns can be readily accessed. These pointers are grouped together in objects called *blocks*, where a single block stores data and index pointers, as well as an array for the diagonal elements, for all the columns associated with a single node of the finite element model. Both the data and index pointers point to the first entry of the columns in the panel arrays, corresponding to the first non-zero entry below the diagonal.

The relationship between the block pointers and the panel arrays is shown in Figure 23. The pointers to the data columns may be widely separated, since each column in the panel requires a distinct segment of storage. By contrast, the block index pointers point to adjacent locations in the panel indices. The pointers are able to share indices in this way because columns belonging to the same block are all associated with the same finite element node and therefore have the same off-diagonal structure. The only difference among them is that the number of off-diagonals for each successive column is reduced by one, which is why the block index pointers are offset from each other by one position.

A sparse matrix object is used to hold and access an array of n block objects, where n is the number of FE nodes. The data and index pointers for all of the block objects are held in memory and maintained throughout the factorization and solution steps regardless which panel is currently loaded in memory. If d is the number of DOFs per node, the memory used by the block objects is $4nd$ for the data pointers, $4nd$ for the index pointers, and $8nd$ for the double-precision diagonal entries (assuming 4 bytes are used for a pointer). This of course is in addition to the memory required for the panel arrays.

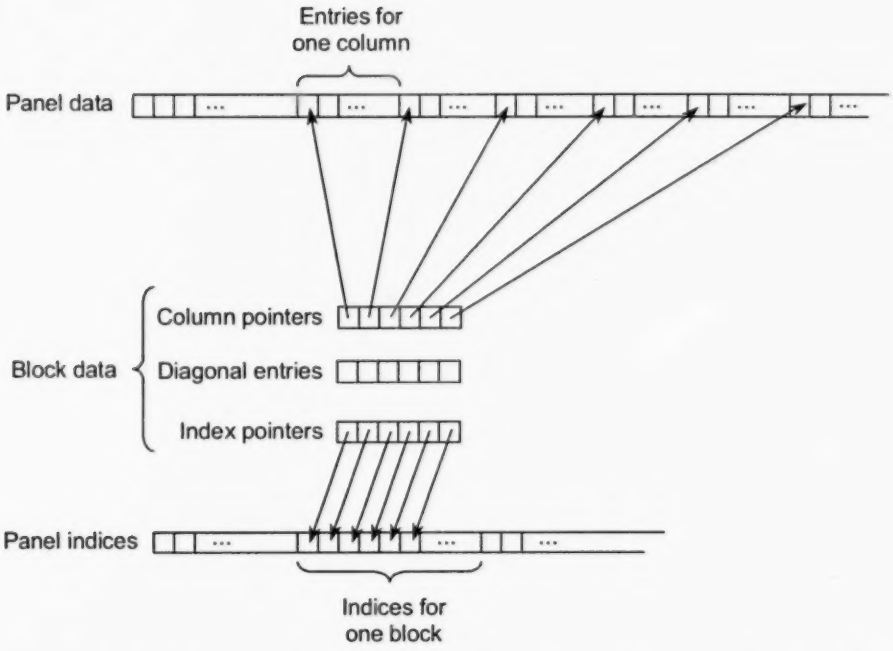


Figure 23: *Block pointers to panel data and index arrays, for a block with six DOF.*

5 Application to Eigenvalue Analysis

The factorization and solution procedures developed for static response analysis can also be used to solve eigenvalue problems of the form

$$Kx = \lambda Bx \tag{3}$$

where λ, x is an eigenvalue/eigenvector pair, and both K and B are sparse, symmetric matrices. K is the global stiffness matrix. In a natural frequency analysis, B is the global mass matrix, while in an elastic buckling mode analysis it is the global "stress stiffening", or "geometric stiffness", matrix.

The power method [12] is an effective method for calculating the largest eigenvalues of a general matrix A. The basic operation of the power method is a vector multiplication:

$$x' = Ax \tag{4}$$

Iterative application of this converges to the eigenvector of the mode with the largest eigenvalue. Additional iterations can be performed to find the second, third modes, etc.

To apply the power method to (3), it must first be transformed to the more restricted form:

$$Ax = \mu x \tag{5}$$

Assuming that K is positive definite and therefore invertible, we define

$$A = K^{-1}B \qquad \mu = 1/\lambda$$

The iteration (4) is therefore carried out by first calculating $y = Bx$, and then solving the linear system $Kx' = y$. The latter solution can be performed using the factorization and solution methods described in the foregoing chapters, assuming a factorization of the form $K = LDL^T$. The power method performed in this way will converge modes corresponding to the largest values of μ, and hence the smallest values of λ.

In natural frequency analysis it is possible that the structure is unconstrained or partially constrained, in which case the stiffness matrix is singular and cannot be factorized. Instead, the eigenvalue problem is solved in the following shifted form

$$(K + \alpha B)x = (\lambda + \alpha)Bx \tag{6}$$

A positive value of α will make the matrix $K + \alpha B$ nonsingular so it can be factorized as described above. The power method can then be applied using $A = (K + \alpha B)^{-1}B$. Note that in the shifted form the eigenvectors are unchanged whereas the eigenvalues are shifted by α.

The factorization is carried out and stored prior to performing the iterations. Each iteration of the power method then requires a matrix-vector product (Bx) and a forward and backward solution. In practice, an additional matrix-vector product is required to normalize the solution vectors to the B matrix.

The factorization and solution steps are implemented as described in Section 4. In natural frequency analysis in which lumped mass modelling is used, the matrix B is diagonal and the product Bx is simple to compute. But in buckling analysis, or in natural frequency analysis in which consistent element mass modelling is used, the matrix-vector products are computed with B in a sparse, symmetric form.

The amount of storage required for sparse, symmetric B is generally much smaller than what is required for L because no factorization, and hence no fill-in, is needed for B. In fact, in large problems, the storage needed for B may be two or three times smaller than what is needed for L.

6 Application to Linear Dynamic Analysis

The factorization and solution methods described in previous chapters can also be used in the solution of time-history dynamic response of structural systems. The solution for this type of analysis can be computed with an implicit time-integration scheme [21]. The solution steps in the scheme are given below, where x, $\dot{x}$ and $\ddot{x}$ are the displacement, velocity, and acceleration vectors, respectively. The displacement vector is calculated by solving the linear system

$$K^{\text{eff}} x_{i+1} = F^{\text{eff}}_{i+1}, \tag{7}$$

where the effective stiffness matrix and effective load vectors are given by the following expressions in which h denotes the time step:

$$K^{\text{eff}} = \frac{4}{h^2} M + \frac{2}{h} C + K \tag{8}$$

$$F^{\text{eff}}_{i+1} = F^{\text{ext}}_{i+1} + M\left(\frac{4}{h^2} x_i + \frac{4}{h}\dot{x}_i + \ddot{x}_i\right) + C\left(\frac{2}{h} x_i + \dot{x}_i\right) \tag{9}$$

The vector F^{ext}_{i+1} holds the external applied loads and C is the damping matrix.

The velocity and acceleration vectors are updated with the following formulas, which are based on the trapezoidal rule:

$$\dot{x}_{i+1} = \frac{2}{h}(x_{i+1} - x_i) - \dot{x}_i \tag{10}$$

$$\ddot{x}_{i+1} = \frac{4}{h^2}(x_{i+1} - x_i) - \frac{4}{h}\dot{x}_i - \ddot{x}_i \tag{11}$$

If proportional damping is assumed such that $C = \alpha_K K + \alpha_M M$, where α_K and α_M are scalar proportionality factors, the equivalent stiffness matrix and equivalent load vector become the following:

$$K^{\text{eff}} = \left(\frac{4}{h^2} + \alpha_M \frac{2}{h}\right) M + \left(1 + \alpha_K \frac{2}{h}\right) K \tag{12}$$

$$F^{\text{eff}}_{i+1} = F^{\text{ext}}_{i+1} + M\left(\frac{2}{h}\left[\frac{2}{h} + \alpha_M\right] x_i + \left[\frac{4}{h} + \alpha_K\right]\dot{x}_i + \ddot{x}_i\right) + \alpha_K K\left(\frac{2}{h} x_i + \dot{x}_i\right) \tag{13}$$

Each iteration in the time integration scheme requires the forward and backward solution of the linear system (7), a matrix-vector product involving the mass matrix M, and, if $\alpha_K \neq 0$, a matrix-vector product involving the stiffness matrix K. The factorization of K_{eff} can be performed prior to the iterations, and therefore only needs to be performed once. Thus the numerical operations required are very similar to those used in the eigenvalue computation, and the same procedures can be used for the forward and backward solution, as well as the matrix-vector products.

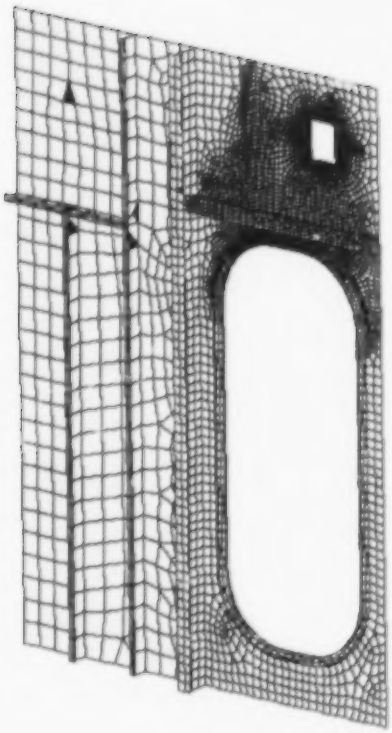


Figure 24: *Longitudinal Bulkhead Detail Model.*

7 Numerical Performance

7.1 Finite Element Models

The eight finite element models that were used to test solver performance of the sparse solver are depicted in Figures 24 to 30. All are models of actual ship or submarine structures or are representative of such structures. The models range in size from 7000 nodes to 60,000 nodes, as shown in Table 1.

Figure 24 shows a longitudinal bulkhead detail from the HALIFAX class frigate. The model is subject to static displacement constraints consistent with an eight-metre static wave balance in hog. An eigenvalue analysis is also performed to find the first ten modes of vibration using a consistent element mass formulation and an eigenvalue tolerance of 10^{-5}.

Figure 25 shows the global structural model of the HALIFAX class frigate. For this model a static analysis was conducted with 207 separate load cases. A still water

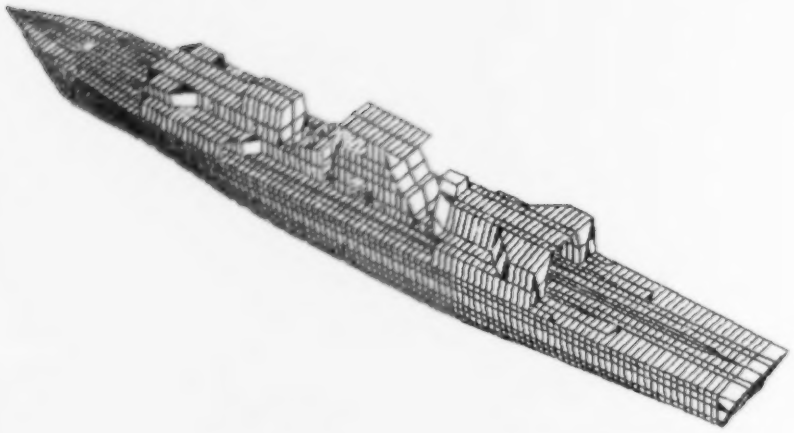


Figure 25: *Global Model of the HALIFAX Class Frigate.*

pressure load, six unit acceleration loads, and 200 unit pressure loads comprise the load cases examined. A natural frequency analysis was also performed on the structure, in which the first thirty modes are calculated assuming a dry structure and lumped mass modelling, and using an eigenvalue tolerance of 10^{-5}.

Figure 26 shows a model of a hypothetical enclosed warship mast. A dynamic response analysis was performed on this mast model with a time-varying pressure applied to the structure representative of an air blast loading. The model was analysed with a consistent mass model for the structural mass, and proportional damping constants of $\alpha_M = .1$ and $\alpha_K = 10^{-5}$. A total of 200 time steps were calculated.

Figure 27 shows a structural detail of the HALIFAX class frigate in way of the helicopter hangar (starboard side). A single load case was applied consistent with an eight-metre static wave balance of the hull in sag. The loading is imposed through displacement constraints on the boundary of the model.

Figure 28 shows a model of a ring-stiffened cylinder typical of submarine hulls. The ends of the cylinder are clamped, and a hydrostatic pressure load is applied. A static analysis is performed on the model, the results of which are used to determine the pre-stressing effect for a subsequent linear buckling analysis. The first twenty linear buckling modes are determined with an eigenvalue tolerance of 10^{-3}.

Figure 29 shows a typical detail model of a ship's hopper structure. Uniform compressive loading was applied to the structure in the longitudinal direction.

Figure 30 shows a detailed model of the keel structure in the KINGSTON class coastal patrol vessel. A static analysis was conducted with a uniform pressure load

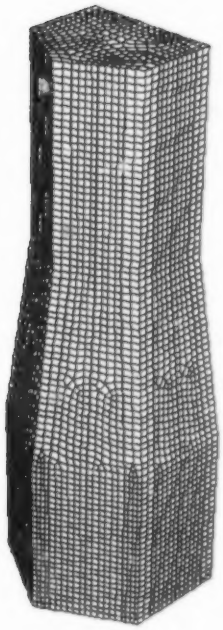


Figure 26: *A model of an enclosed warship mast.*

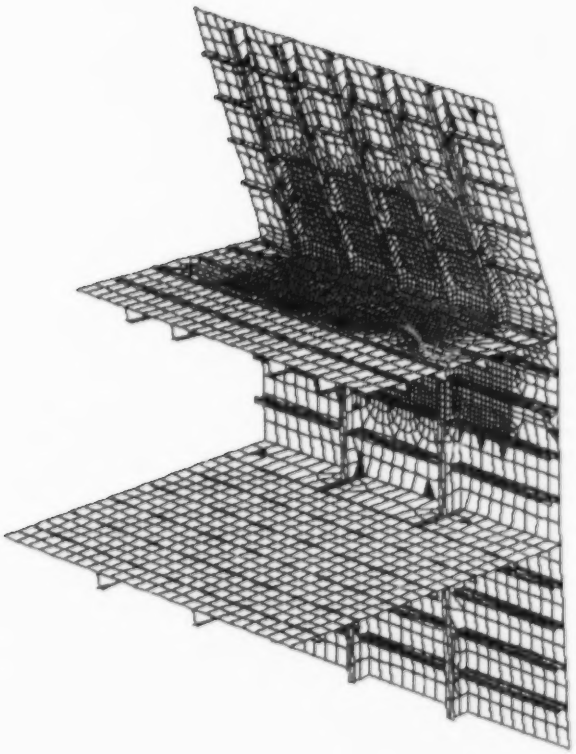


Figure 27: *HALIFAX Class Side Shell Structure in Way of Hangar.*

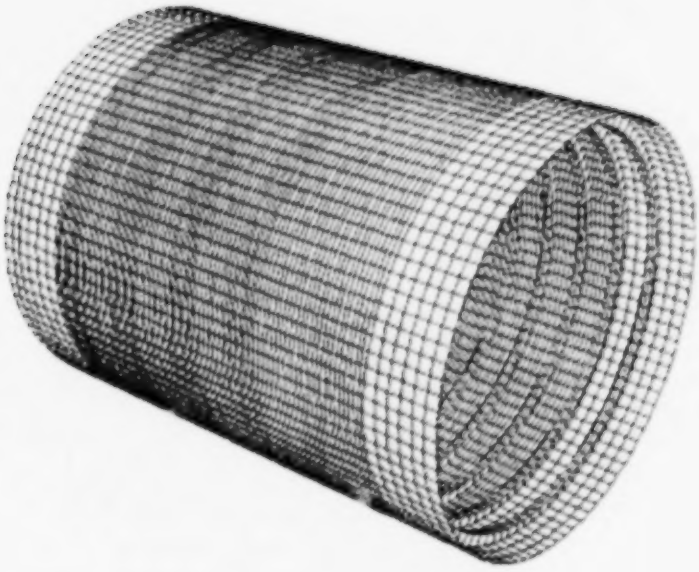


Figure 28: *Ring-Stiffened Cylinder Model.*

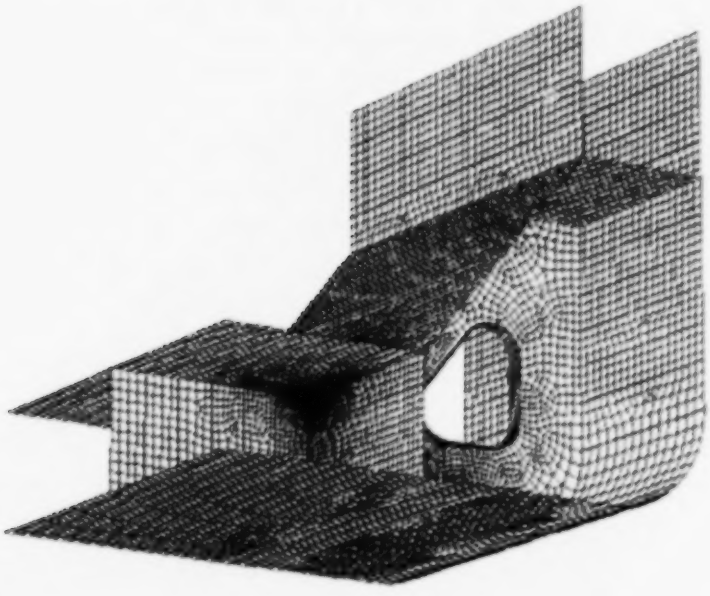


Figure 29: *Detail Model of a Ship's Hopper.*

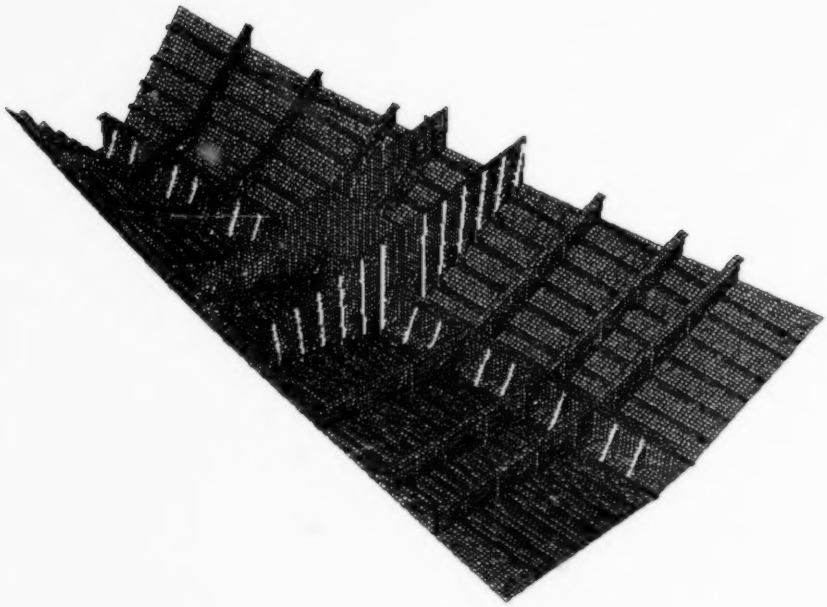


Figure 30: *KINGSTON Class Keel Structure.*

applied to the hull plating and with boundaries fixed. A dynamic response analysis was also performed using a uniform, time varying pressure with a peak value of 140 kPa, which might represent a typical slamming load case. The model was analysed with a lumped mass model for the structural mass, and proportional damping constants of $\alpha_M = 0.1$ and $\alpha_K = 10^{-5}$. A total of 200 time steps were calculated.

Figure 31 is a detail of the centerline bulkhead structure in the IROQUOIS class destroyer. A single static load case is analysed, consisting of a uniform tensile load applied to the hull plating in the longitudinal direction.

7.2 Storage Requirements

The MD and GND ordering methods were tested on the eight finite element models, and the resulting storage requirements for the factorized stiffness matrix were recorded (Table 2). The numbers in the table denote the total size of the factorized matrix (data and indices combined). In Section 3.3 it was noted that a tie-breaking rule is needed for the MD method when multiple vertices are found to have minimum degree. In Table 2, MD results are presented for three different tie-breaking rules in which the minimum degree vertex is selected as the following: the first-encountered minimum degree vertex (MD-1); the last-encountered minimum de-

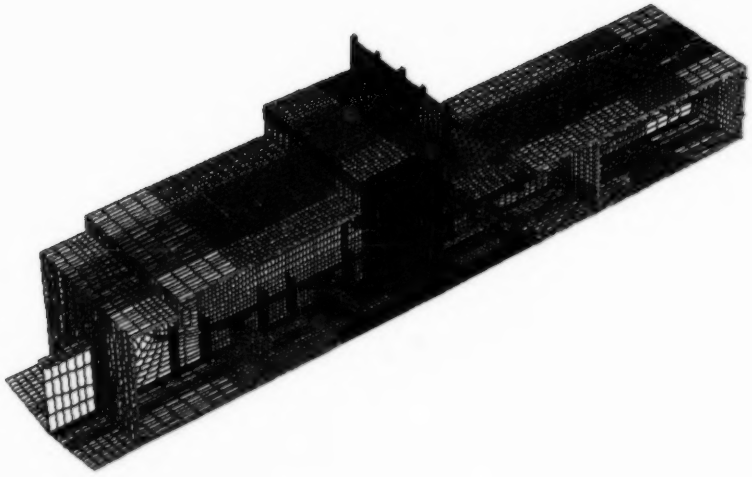


Figure 31: *IROQUOIS Centerline Bulkhead Detail Model.*

Table 1: *Summary of Example Models*

Model	Nodes	DOF	Elements
Longitudinal bulkhead	7,160	42,960	7,146
HALIFAX global model	7,534	45,204	17,184
Enclosed mast	8,270	49,620	9,851
Hangar detail	10,645	63,870	10,911
Stiffened cylinder	12,672	76,032	12,576
Hopper detail	25,154	150,924	25,170
KINGSTON keel	41,585	249,510	43,039
Centerline bulkhead	62,056	372,336	64,334

***Table 2:** Matrix Size (in MB) Produced by Ordering Methods*

Model	MD-1	MD-L	MD-R	GND
Longitudinal bulkhead	61.83	61.17	59.59	88.66
HALIFAX global model	108.98	108.77	109.69	158.48
Enclosed mast	93.96	91.84	103.78	109.91
Hangar	87.80	87.33	89.29	162.47
Cylinder	112.60	113.32	134.39	174.32
Hopper	291.59	282.41	287.76	729.73
KINGSTON keel	456.40	450.54	460.11	875.38
Centerline bulkhead	945.22	857.23	932.18	1633.93

gree vertex (MD-L); and a randomly selected minimum degree vertex (MD-R). In the case of the MD-R, a seeded random number generator is used so that the particular ordering for a given model will vary from one execution to another. The results given for MD-R represent the average matrix storage over ten executions.

The matrix storage results in Table 2 clearly indicate the superiority of MD over GND. MD-L gives the best ordering (i.e., smallest matrix storage) in six of the eight models. Only in the longitudinal bulkhead structure does MD-R give a slightly better ordering, while in the cylinder model MD-1 is slightly better. In most of the problems tested little difference is observed for the different tie-breaking rules; the exception is the centerline bulkhead where MD-L is notably better than the other two. Based on these results, MD-L was used for all subsequent analyses results presented here involving the sparse solver.

It may be a coincidence that MD-L produces the best ordering for most of the models. Another possibility is that the original mesh generation orders the nodes in a way that somehow favours MD-L, but it is not understood how this would occur.

7.3 Static Solution Performance

Static response analysis was performed for seven of the models. The results were computed using both the sparse and profile solvers, and in all cases, identical displacement and stress solutions were obtained with the two solvers. The execution times given in Figure 3 were measured on a Pentium 4 desktop PC with a 1.8 GHz processor and 1 GB of RAM. The times are broken into two program steps: the assembly and factorization (AF) step and the solution (S) step.

From these results it is clear that the sparse solver provides greatly improved performance over the profile solver. The increase in speed varies from 2.1 times, in

***Table 3:** VAST85 Static Analysis Execution Times (seconds)*

Model	Program Step	Sparse	Profile
Longitudinal bulkhead	AF	14.92	113.77
(1 load case)	S	0.38	4.55
	Total	15.30	118.32
HALIFAX global	AF	44.36	70.89
(207 load cases)	S	54.69	146.36
	Total	99.05	217.25
Hangar	AF	30.00	425.64
(1 load case)	S	0.94	10.55
	Total	30.94	436.19
Stiffened cylinder	AF	35.58	377.61
(1 load case)	S	0.81	10.91
	Total	36.39	388.52
Hopper	AF	152.22	6890.70
(1 load case)	S	1.11	210.70
	Total	153.33	7101.40
Keel	AF	333.78	3596.30
(1 load case)	S	2.83	181.70
	Total	336.61	3778.00
Centerline bulkhead	AF	1667.41	21,537.98
(1 load case)	S	49.27	693.97
	Total	1716.68	22,231.95

the HALIFAX global model, to 46 times, in the hopper detail model. Overall, the improved performance can be attributed to two factors. One is the more efficient representation of the factorized stiffness matrix in the sparse solver, which leads to smaller storage requirement and fewer operations. The other factor is that the sparse solver uses dynamic memory allocation, and can therefore use all of the available system memory; whereas the profile solver is limited to a fixed array size of 40 MB.

Some other observations may be made on the performance statistics. Relative to the profile solver, the sparse solver is generally better in the detail models than in the HALIFAX global model. The reason for this lies in the differences among the mesh geometries. Detail models like the hopper and hangar do not exhibit any slenderness, i.e., the number of nodes is large in every direction. Regardless of the ordering, the bandwidth of this type of model is necessarily large and profile methods will therefore perform poorly.

The HALIFAX global model on the other hand is distinctly slender in two of its dimensions, and the profile method can be effective in this situation. A further point is that the HALIFAX global model has a very large number of elements for the number of nodes. The assembled matrix therefore has relatively more coupling terms, and is therefore less sparse than for the other problems tested. With less sparsity in the matrix to exploit, the sparse factorization method cannot be expected to perform as well. The sparse solver nonetheless outperforms the profile solver in this problem.

With 1 GB of available memory, all but one of the models were analysed using a single-panel representation of the factorized matrix. Thus, these models could all be analysed without any external storage of the factorized matrix. The exception was the centerline bulkhead model, which required a two-panel representation of the factorized matrix.

7.4 Eigenvalue Analysis Performance

Eigenvalue analysis was conducted on three of the models. In two cases (the longitudinal bulkhead and the HALIFAX global model) natural frequency analysis was performed; in the third (ring-stiffened cylinder model) linear buckling analysis was performed. Identical solutions for the eigenvalues and eigenvectors were obtained with the two solvers in two natural frequency analysis cases. In the buckling analysis the solutions were not quite identical, but for practical purposes the differences were insignificant. The reason identical solutions are not obtained has to do with the starting vectors used to begin the power iterations. These vectors are not necessarily the same for the two solvers, and as a result the buckling modes can converge in a different manner.

Table 4: *VAST85 Eigenvalue Analysis Execution Times (seconds)*

Model	Program Step	Sparse	Profile
Bulkhead	AF	14.67	128.80
(10 modes)	S	59.55	1869.47
	Total	74.22	1998.27
HALIFAX Global	AF	51.45	114.13
(30 modes)	S	483.53	4759.97
	Total	543.98	4874.10
Stiffened Cylinder	AF	36.56	666.97
(20 modes)	S	149.22	18,342.70
	Total	185.78	19,009.67

The performance of the sparse solver in eigenvalue analysis is superb, as can be seen from the execution times given in Table 4. The performance margin observed in static analysis problems is even larger in eigenvalue analysis. This can especially be seen for the ring-stiffened cylinder problem, where the sparse solver is a little over ten times faster in static analysis, but is over one hundred times faster in the buckling analysis. This is a direct result of the greater efficiency with which forward and backward solution and matrix-vector multiplication can be performed in a sparse format.

7.5 Dynamic Analysis Performance

Transient dynamic analysis was conducted for two of the models (warship mast and the keel model). Identical displacement, velocity and acceleration solutions were obtained with the two solvers for both of the structures. The execution times for the two models are given in Table 5. The sparse solver outperforms the profile solver by more than a factor of ten in the enclosed mast problem, and by over a factor of one hundred in the keel problem.

The sparse solver performs extremely well in dynamic analysis problems, largely due to the high efficiency achieved by using a sparse format in the time integration steps. Both the backward and forward solution steps and the matrix-vector products that are performed during each iteration of the time integration are carried out much more quickly than in the profile solver. This performance margin should increase further with (1) increased model sizes, (2) increased numbers of time steps, and (3) increased physical memory.

Table 5: *VAST85 Dynamic Analysis Execution Times (seconds)*

Model	Program Step	Sparse	Profile
Mast	AF	31.45	302.53
(200 steps)	S	92.02	6760.16
	Total	123.47	7062.69
Keel	AF	330.97	4252.39
(200 steps)	S	446.66	85,348.53
	Total	777.63	89,600.92

8 Conclusion

This Memorandum has described the development and implementation of a sparse direct solver in the VAST finite element program. The solver has been found to give very good performance in static, eigenvalue and dynamic response analysis over a variety of ship structural models. Based on the execution times that have been obtained, the VAST sparse solver should satisfy the IST's requirements for analysis of global models, top-down local models and details. In this respect, the sparse solver development and implementation work described in this report can be regarded as completed. Furthermore, in a previous study the sparse solver was found to give a level of performance in static analysis comparable to the commercial finite element analysis software ANSYS [7].

One capability not yet implemented in the sparse solver is the inclusion of fluid added-mass in natural frequency and dynamic response analysis. While this is an important capability for ship structures, it is not anticipated that the sparse solver will be effective for problems with fluid-added mass. This is because fluid-added mass matrices tend to be large in size and dense (fully-coupled) and thus are lacking in sparsity that can be exploited.

If necessary, further improvements in the sparse solver may be achieved with better node ordering algorithms. Although generalised nested dissection was found to be inferior to the minimum degree algorithm, it is possible that more advanced nested dissection methods have been developed that may be superior. For example, Henrickson and Rothberg [22] have described a number of improvements to nested dissection that may be worth investigating.

References

1. Martec Limited. (1997). VAST – Vibration and Strength Analysis. Version 7.2 User's Manual.

2. Pegg, N. G. (1999). Development of the ISSMM Software Tool for Residual Strength Assessment of Ship Structures. (DREA TM 1999-055). Defence Research Establishment Atlantic.

3. Jiang, L., Norwood, M. E., Chernuka, M. W., and Covill, J. D. (1996). Pilot Study in Improving Efficiency of the VAST Finite Element Program. (TR-96-19). Martec Technical Report.

4. Jiang, L., and Chernuka, M. W. (1998). IVAST-An Efficient VAST Finite Element Program for Linear and Nonlinear Static Analysis. (TR-98-05) Martec Technical Report.

5. Jiang, L., and Chernuka, M. (2000). Improved Nonlinear Solution Techniques in the VAST Finite Element Program. (DREA CR 2000-101). Defence Research Establishment Atlantic.

6. Smith, M. J. (2001). First-Ply Failure Prediction of Fibrous Composites with Finite Element Analysis. (DREA TM 2001-065). Defence Research Establishment Atlantic.

7. ANSYS Inc. (2002). ANSYS Software and Documentation (Release 6.1).

8. George, A., and Liu, J. W-H. (1981). Computer Solution of Large Sparse Positive Definite Systems. Englewood Cliffs, NJ: Prentice-Hall.

9. Pissanetsky, S. (1984). Sparse Matrix Technology. London: Academic Press.

10. Axelsson, O., and Barker, V. A. (1984). Finite Element Solution of Boundary Value Problems. London: Academic Press.

11. Golub, G. H., and Van Loan, C. F. (1989). Matrix Computations. 2nd Ed. Baltimore: Johns Hopkins University Press.

12. Hager, W. W. (1988). Applied Numerical Linear Algebra. Englewood Cliffs: Prentice-Hall.

13. Strang, G. (1980). Linear Algebra and its Applications. 2nd Ed. Orlando, FA: Academic Press.

14. Rose, D. J. (1972). A Graph Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations. In R. C. Read (Ed.) *Graph Theory and Computation*, pp. 183–217. London: Academic Press.

15. George, A., and Liu, J. W-H. (1989). The Evolution of the Minimum Degree Ordering Algorithm. SIAM Review, 31, 1, 1–19.

16. Stroustrup, B. (1997). The C++ Programming Language. 3rd Ed. Reading, Massachusetts: Addison-Wesley.

17. Ng, E. G., and Peyton, B. W. (1993). Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers. *SIAM Journal of Scientific Computing*, 14(5) 1034–1056.

18. Rothberg, E., and Schreiber, R. (1999). Efficient Methods for Out-of-Core Sparse Cholesky Factorization. *SIAM Journal of Scientific Computing*, 21(1) 129–144.

19. Ashcraft, C., and Liu, J. W. H. (1998). Robust Ordering of Sparse Matrix Using Multisection. SIAM Journal of Matrix Analysis Applications, 19(3) 816-832.

20. Jiang, L. (2002). Implementation of a Sparse Direct Matrix Solver for the VAST Finite Element Program. (DREA CR 2002-026). DRDC Atlantic.

21. Cook, R. D., Malkus, D. S., and Plesha, M. E. (1989). Concepts and Applications of Finite Element Analysis, Third Edition. New York: John Wiley & Sons.

22. Hendrickson, B., and Rothberg, E. (1998). Improving the Run Time and Quality of Nested Dissection Ordering. *SIAM Journal of Scientific Computing*, 20 (2) 468–489.

Annex A
List of symbols, abbreviations, acronyms and initialisms

AF	assembly and factorization
DOF	degree of freedom
DSA	detail stress analysis
FEA	finite element analysis
FELAM	Finite Element Analysis of Laminates
GND	generalized nested dissection
ISSMM	Improved Ship Structure Maintenance Management
IST	ISSMM Software Tool
MAESTRO	Method for Analysis Evaluation and Structural Optimization
MD	minimum degree
MD-1	minimum degree algorithm using first-encountered MD vertex tie breaking strategy
MD-L	minimum degree algorithm using last-encountered MD vertex tie breaking strategy
MD-R	minimum degree algorithm using random MD vertex tie breaking strategy
S	solution
VAST	Vibration and Strength Analysis

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence R&D Canada - Atlantic
PO Box 1012
Dartmouth, NS, Canada B2Y 3Z7

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable).

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title).

A Sparse Solver for VAST

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

Smith, Malcolm, J.

5. DATE OF PUBLICATION (month and year of publication of document)

September 2002

6a. NO. OF PAGES (total containing information Include Annexes, Appendices, etc).

55 (approx.)

6b. NO. OF REFS (total cited in document)

22

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered).

Technical Memorandum

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include address).

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant).

1.g.c.11

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written).

10a ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.)

DRDC(A) TM-2002-140

10b OTHER DOCUMENT NOs. (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Defence departments and defence contractors; further distribution only as approved
( ) Defence departments and Canadian defence contractors; further distribution only as approved
( ) Government departments and agencies; further distribution only as approved
( ) Defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected).

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

A sparse direct symmetric equation solver has been developed and implemented in DRDC's VAST finite element program for linear and nonlinear static analysis, eigenvalue analysis and time-history dynamic analysis. A review of sparse direct solution procedures is provided, including a discussion of fill reducing node ordering procedures. Details are given describing the sparse factorization and solution procedures implemented for very large systems of equations. The extension of the equation solver to eigenvalue and dynamic response analysis is also described. The performance of the sparse solver is compared to VAST's profile solver using several ship structure finite element models ranging in size from 7,000 to 60,000 nodes. In static response analysis the execution times for the sparse solver are typically an order of magnitude smaller than those obtained with the profile solver. In eigenvalue and dynamic response analysis the time savings are even greater. The minimum degree algorithm is found to be more effective at reducing the fill than is the generalized nested dissection method.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

Ship structure
Finite element analysis
Sparse factorization
Minimum degree algorithm
Nested dissection
Out-of-core solution method

Defence R&D Canada

Canada's leader in defence and national security R&D

R & D pour la défense Canada

Chef de file au Canada en R & D pour la défense et la sécurité nationale

www.drdc-rddc.gc.ca